

---

# Overview of Computer Organization

Chapter 1  
S. Dandamudi

---

# Outline

---

- Introduction
  - \* Basic Terminology and Notation
- Views of computer systems
- User's view
- Programmer's view
  - \* Advantages of high-level languages
  - \* Why program in assembly language?
- Architect's view
- Implementer's view
- Processor
  - \* Execution cycle
  - \* Pipelining
  - \* RSIC and CISC
- Memory
  - \* Basic memory operations
  - \* Design issues
- Input/Output
- Interconnection: The glue
- Historical Perspective
- Technological Advances

# Introduction

---

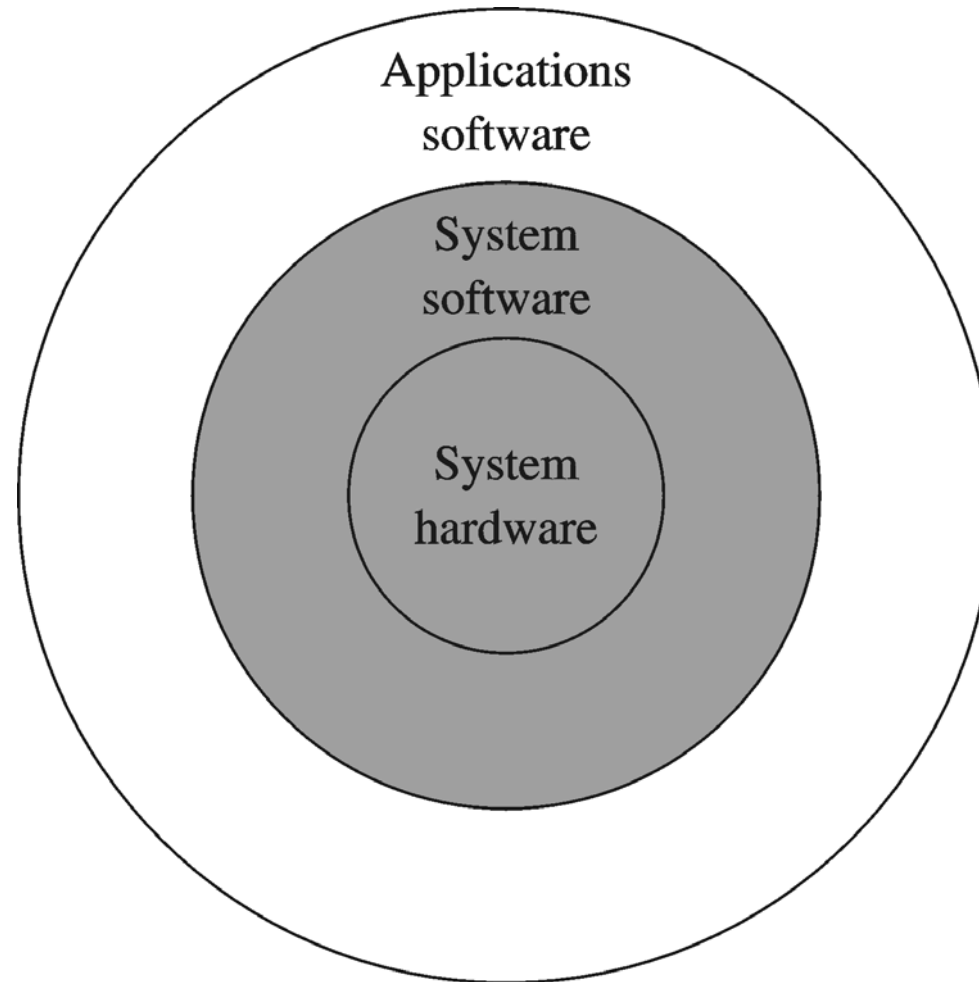
- Some basic terms
  - \* Computer architecture
  - \* Computer organization
  - \* Computer design
  - \* Computer programming
- Various views of computer systems
  - \* User's view
  - \* Programmer's view
  - \* Architect's view
  - \* Implementer's view

## Introduction (cont'd)

<b>Term</b>	<b>Decimal</b>	<b>Binary</b>
K (kilo)	$10^3$	$2^{10}$
M (mega)	$10^6$	$2^{20}$
G (giga)	$10^9$	$2^{30}$
T (tera)	$10^{12}$	$2^{40}$
P (peta)	$10^{15}$	$2^{50}$


# A User's View of Computer Systems

---



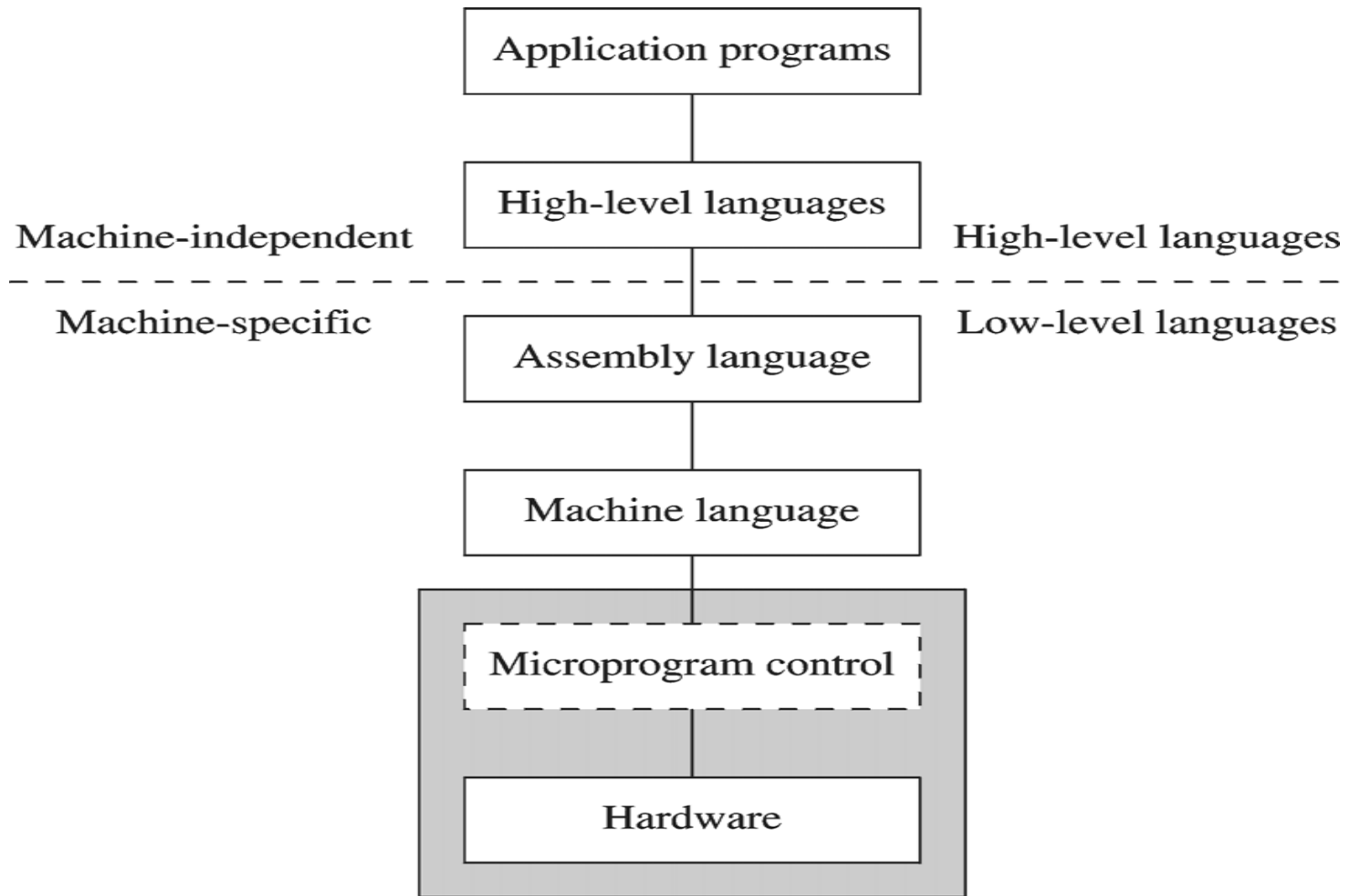
# A Programmer's View

---

- Depends on the type and level of language used
  - A hierarchy of languages
    - \* Machine language
    - \* Assembly language
    - \* High-level language
    - \* Application programs
  - Machine-independent
    - \* High-level languages/application programs
  - Machine-specific
    - \* Machine and assembly languages
- 
- increasing level  
of abstraction

# A Programmer's View (cont'd)

---



## A Programmer's View (cont'd)

---

- Machine language

- \* Native to a processor

- \* Consists of alphabet 1s and 0s

**1111 1111 0000 0110 0000 1010 0000 0000B**

- Assembly language

- \* Slightly higher-level language

- \* Human-readable

- \* One-to-one correspondence with most machine language instructions

**inc          count**



## A Programmer's View (cont'd)

---

- Readability of assembly language instructions is much better than the machine language instructions
  - » Machine language instructions are a sequence of 1s and 0s

<b>Assembly Language</b>	<b>Machine Language (in Hex)</b>
<b>inc      result</b>	<b>FF060A00</b>
<b>mov      class_size, 45</b>	<b>C7060C002D00</b>
<b>and      mask, 128</b>	<b>80260E0080</b>
<b>add      marks, 10</b>	<b>83060F000A</b>

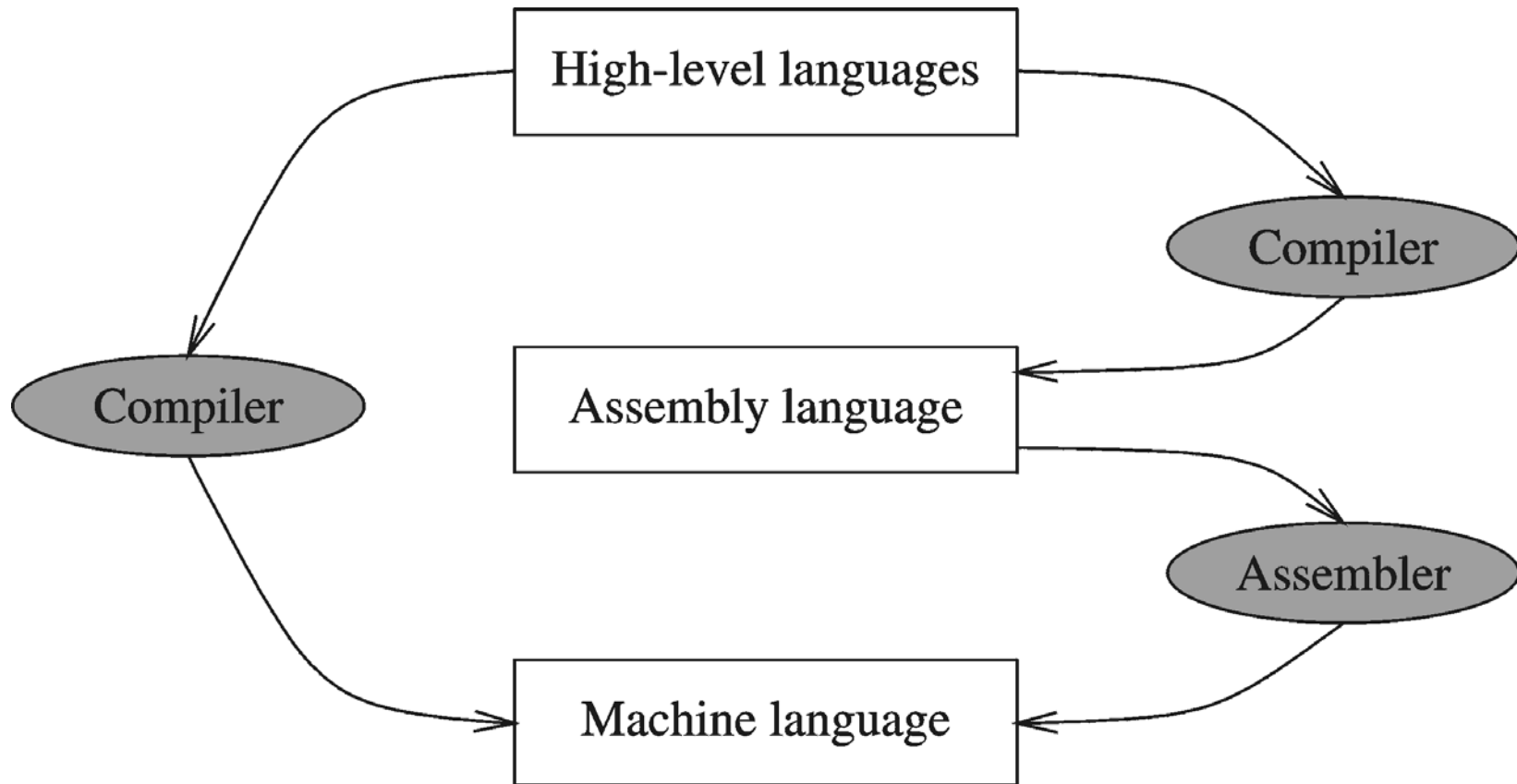
## A Programmer's View (cont'd)

---

- Assemblers translate between assembly and machine languages
  - \* TASM
  - \* MASM
  - \* NASM
- Compiler translates from a high-level language to machine language
  - \* Directly
  - \* Indirectly via assembly language

# A Programmer's View (cont'd)

---



## A Programmer's View (cont'd)

---

- High-level languages versus low-level languages

In C:

```
result =
```

```
count1 + count2 + count3 + count4
```

In Pentium assembly language:

```
mov    AX, count1  
add    AX, count2  
add    AX, count3  
add    AX, count4  
mov    result, AX
```

## A Programmer's View (cont'd)

---

- Some simple high-level language instructions can be expressed by a single assembly instruction

<b>Assembly Language</b>	<b>C</b>
<code>inc result</code>	<code>result++;</code>
<code>mov size, 45</code>	<code>size = 45;</code>
<code>and mask1, 128</code>	<code>mask1 &amp;= 128;</code>
<code>add marks, 10</code>	<code>marks += 10;</code>

## A Programmer's View (cont'd)

---

- Most high-level language instructions need more than one assembly instruction

<b>C</b>	<b>Assembly Language</b>
<code>size = value;</code>	<code>mov AX, value</code> <code>mov size, AX</code>
<code>sum += x + y + z;</code>	<code>mov AX, sum</code> <code>add AX, x</code> <code>add AX, y</code> <code>add AX, z</code> <code>mov sum, AX</code>

# A Programmer's View (cont'd)

---

- Instruction set architecture (ISA)
  - \* An important level of abstraction
  - \* Specifies how a processor functions
    - » Defines a logical processor
- Various physical implementations are possible
  - \* All logically look the same
  - \* Different implementations may differ in
    - » Performance
    - » Price
- Two popular examples of ISA specifications
  - \* SPARC and JVM

# Advantages of High-Level Languages

---

- Program development is faster
  - » High-level instructions
    - Fewer instructions to code
- Program maintenance is easier
  - » For the same reasons as above
- Programs are portable
  - » Contain few machine-dependent details
    - Can be used with little or no modifications on different types of machines
  - » Compiler translates to the target machine language
  - » Assembly language programs are not portable



# Why Program in Assembly Language?

---

- Two main reasons:
  - \* Efficiency
    - » Space-efficiency
    - » Time-efficiency
  - \* Accessibility to system hardware
- Space-efficiency
  - \* Assembly code tends to be compact
- Time-efficiency
  - \* Assembly language programs tend to run faster
    - » Only a well-written assembly language program runs faster
      - Easy to write an assembly program that runs slower than its high-level language equivalent

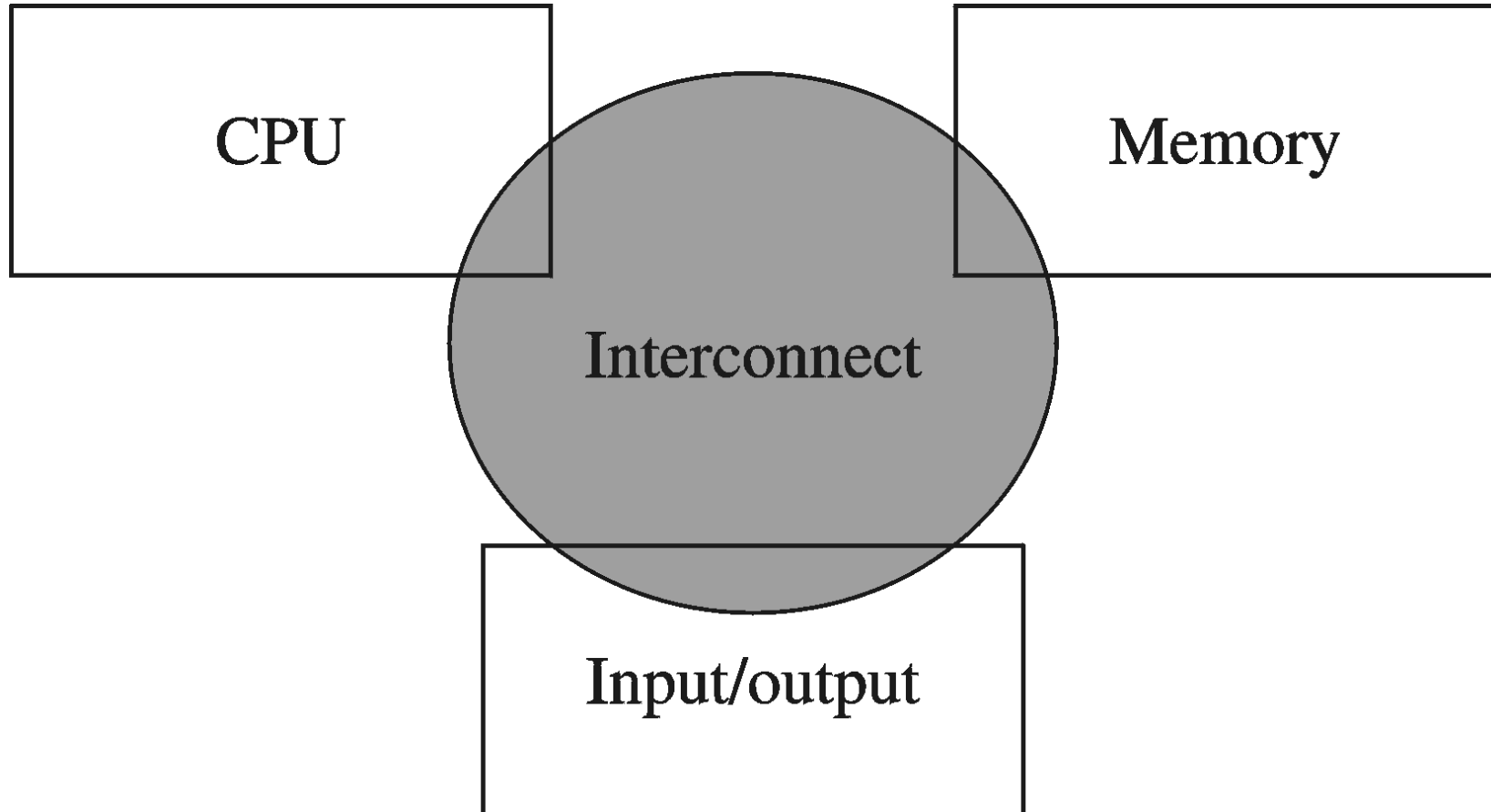
# Architect's View

---

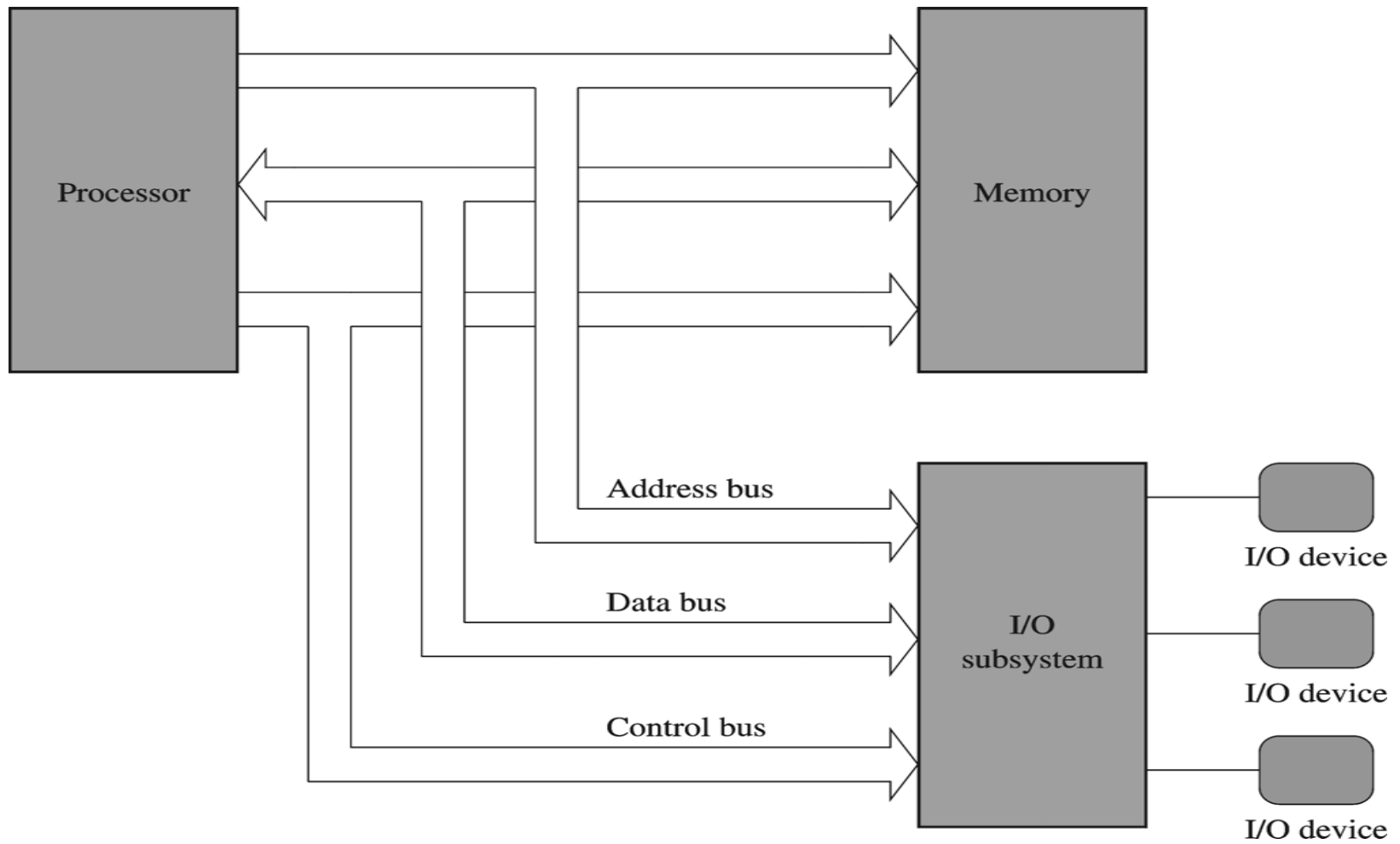
- Looks at the design aspect from a high level
  - \* Much like a building architect
  - \* Does not focus on low level details
  - \* Uses higher-level building blocks
    - » Ex: Arithmetic and logical unit (ALU)
- Consists of three main components
  - \* Processor
  - \* Memory
  - \* I/O devices
- Glued together by an interconnect

## Architect's View (cont'd)

---



# Architect's View (cont'd)

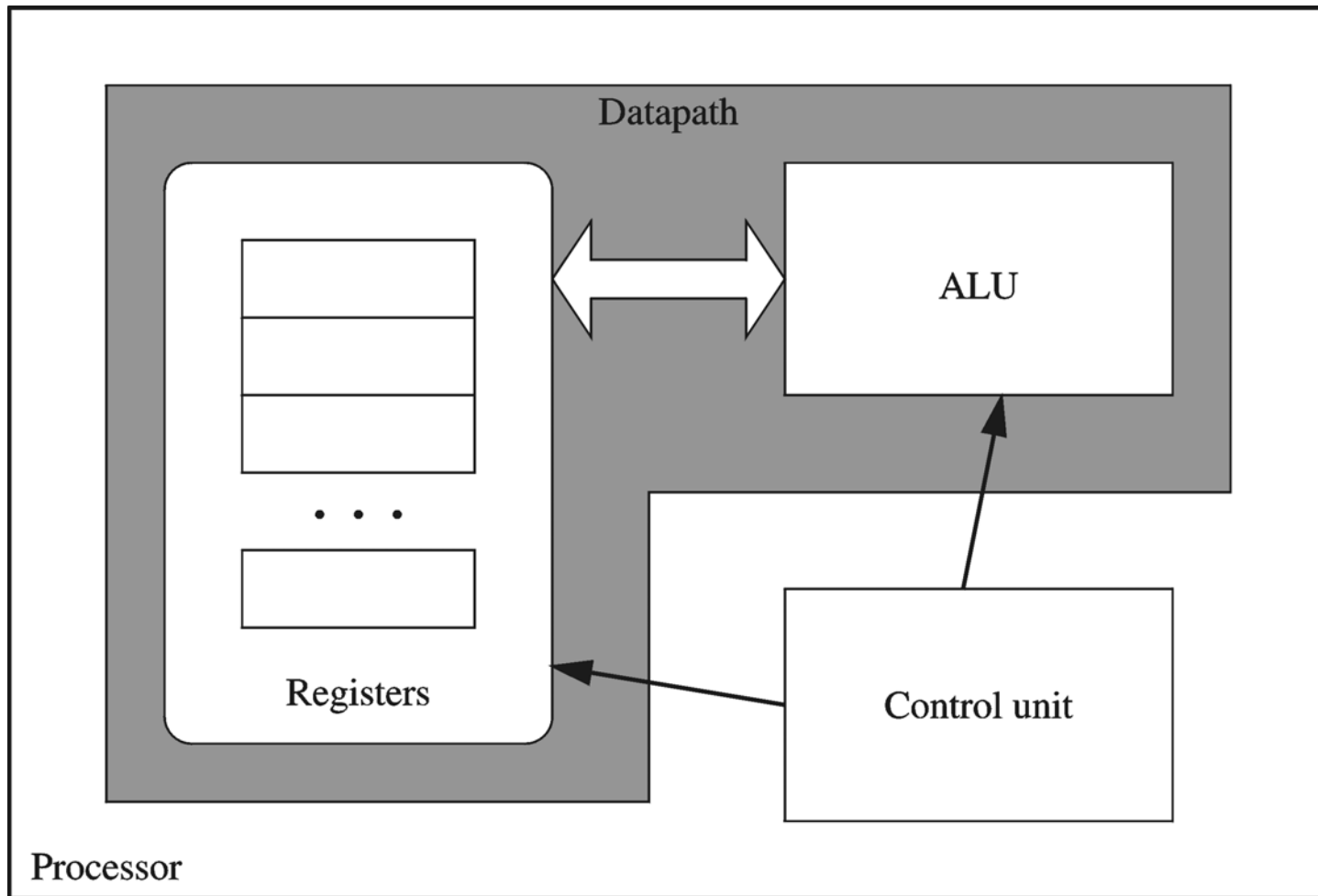


# Implementer's View

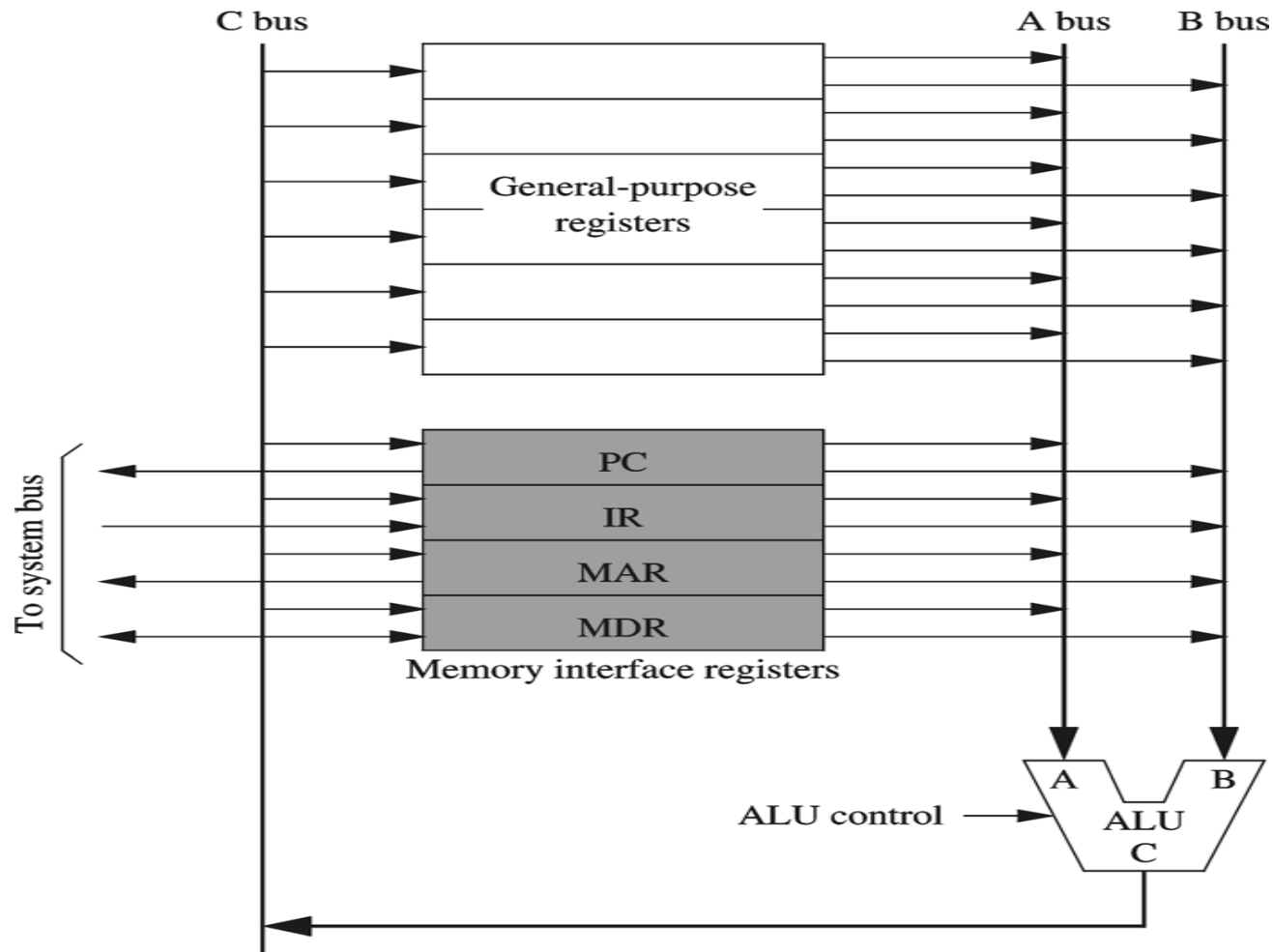
---

- Implements the designs generated by architects
  - \* Uses digital logic gates and other hardware circuits
- Example
  - \* Processor consists of
    - » Control unit
    - » Datapath
      - ALU
      - Registers
- Implementers are concerned with design of these components

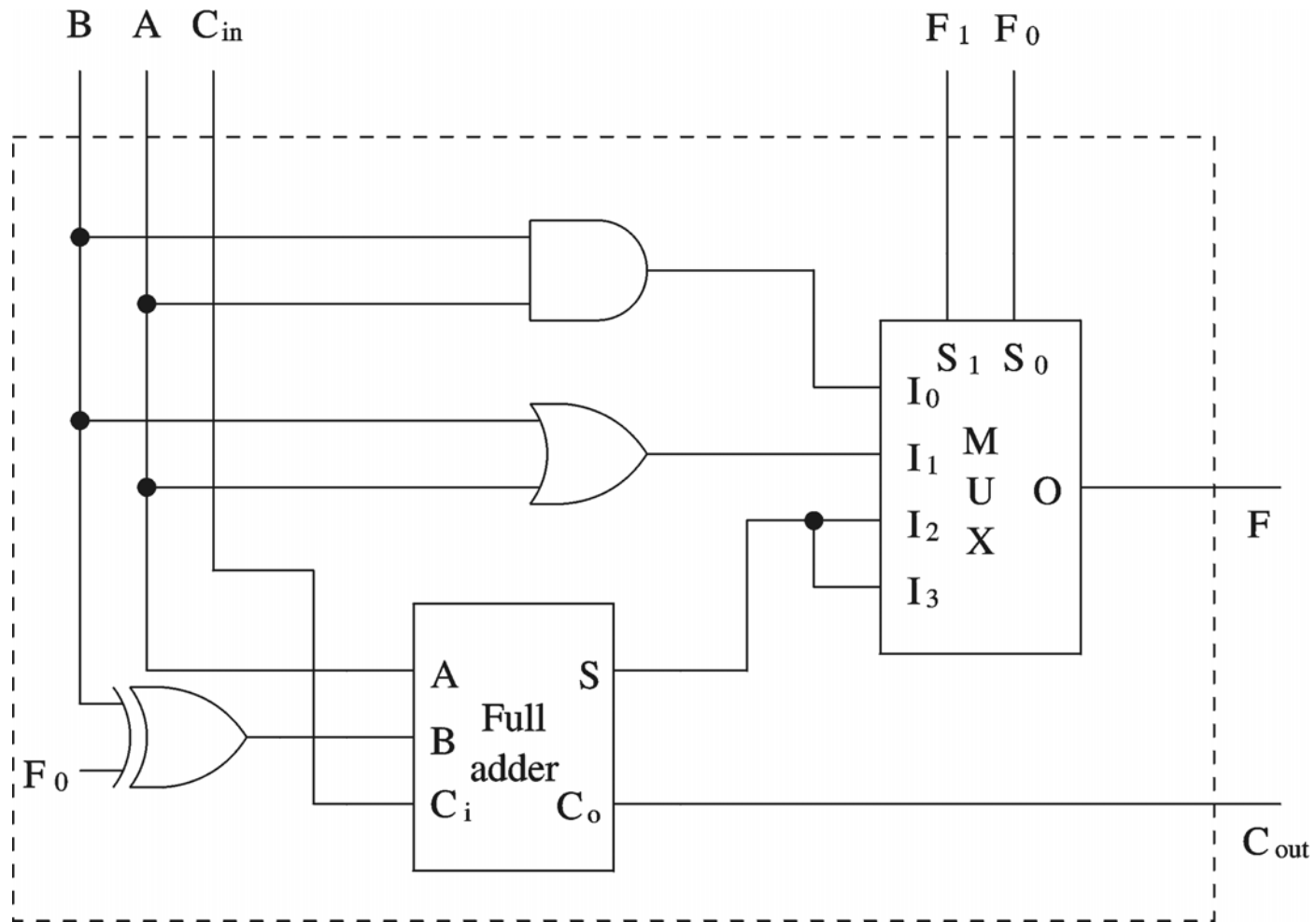
# Implementer's View (cont'd)



# Implementer's View (cont'd)



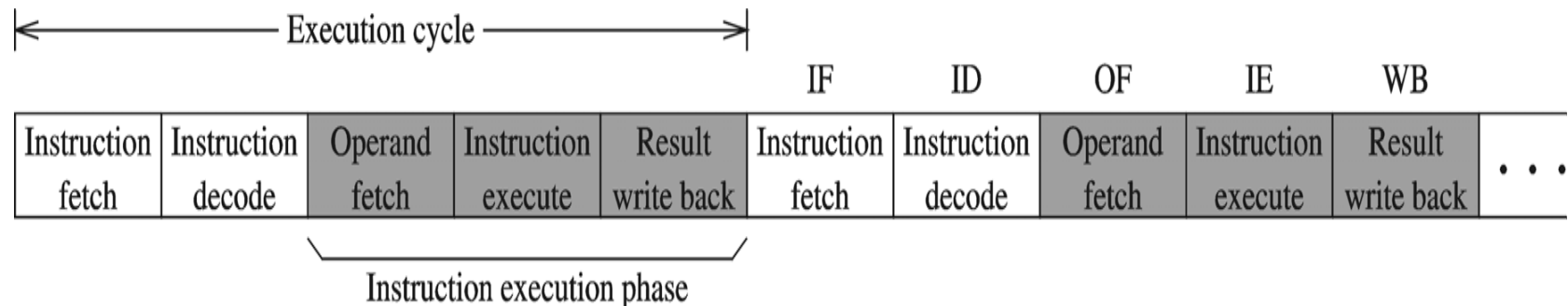
# Implementer's View (cont'd)





# Processor

- Execution cycle
  - Fetch
  - Decode
  - Execute
- von Neumann architecture
  - » Stored program model
    - No distinction between data and instructions
    - Instructions are executed sequentially



# Processor (cont'd)

- Pipelining
  - \* Overlapped execution
  - \* Increases throughput

Time (cycles)  $\longrightarrow$

Stage	1	2	3	4	5	6	7	8	9	10
S1: IF	I1	I2	I3	I4	I5	I6	• • •			
S2: ID		I1	I2	I3	I4	I5	I6	• • •		
S3: OF			I1	I2	I3	I4	I5	I6	• • •	
S4: IE				I1	I2	I3	I4	I5	I6	• •
S5: WB					I1	I2	I3	I4	I5	I6

# Processor (cont'd)

- Another way of looking at pipelined execution

Time (cycles) →

Instruction	1	2	3	4	5	6	7	8	9	10
I1	IF	ID	OF	IE	WB					
I2		IF	ID	OF	IE	WB				
I3			IF	ID	OF	IE	WB			
I4				IF	ID	OF	IE	WB		
I5					IF	ID	OF	IE	WB	
I6						IF	ID	OF	IE	WB

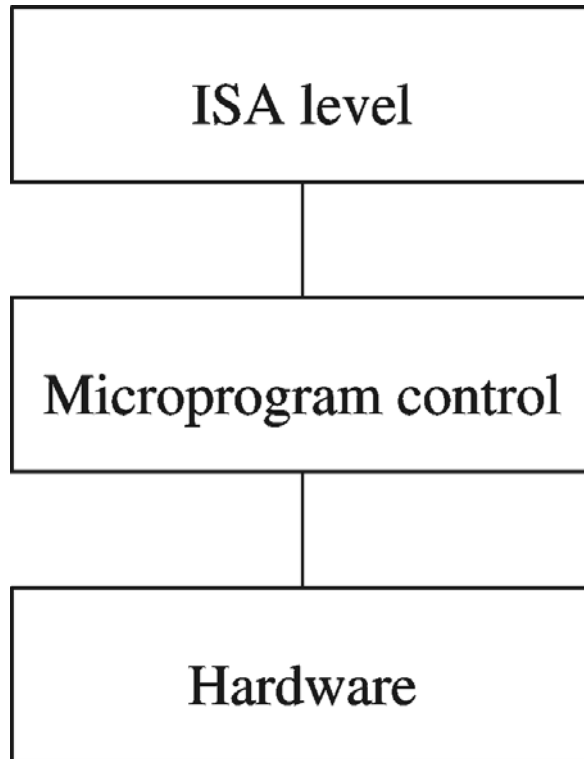
# Processor (cont'd)

---

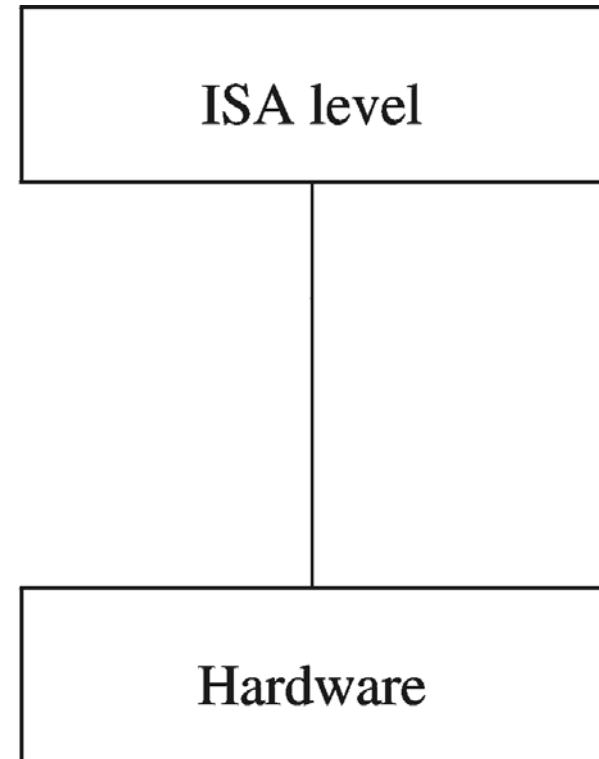
- RISC and CISC designs
  - \* Reduced Instruction Set Computer
    - » Uses simple instructions
    - » Operands are assumed to be in processor registers
      - Not in memory
      - Simplifies design
    - Example: Fixed instruction size
  - \* Complex Instruction Set Computer
    - » Uses complex instructions
    - » Operands can be in registers or memory
      - Instruction size varies
    - » Typically uses a microprogram

## Processor (cont'd)

---



(a) CISC implementation

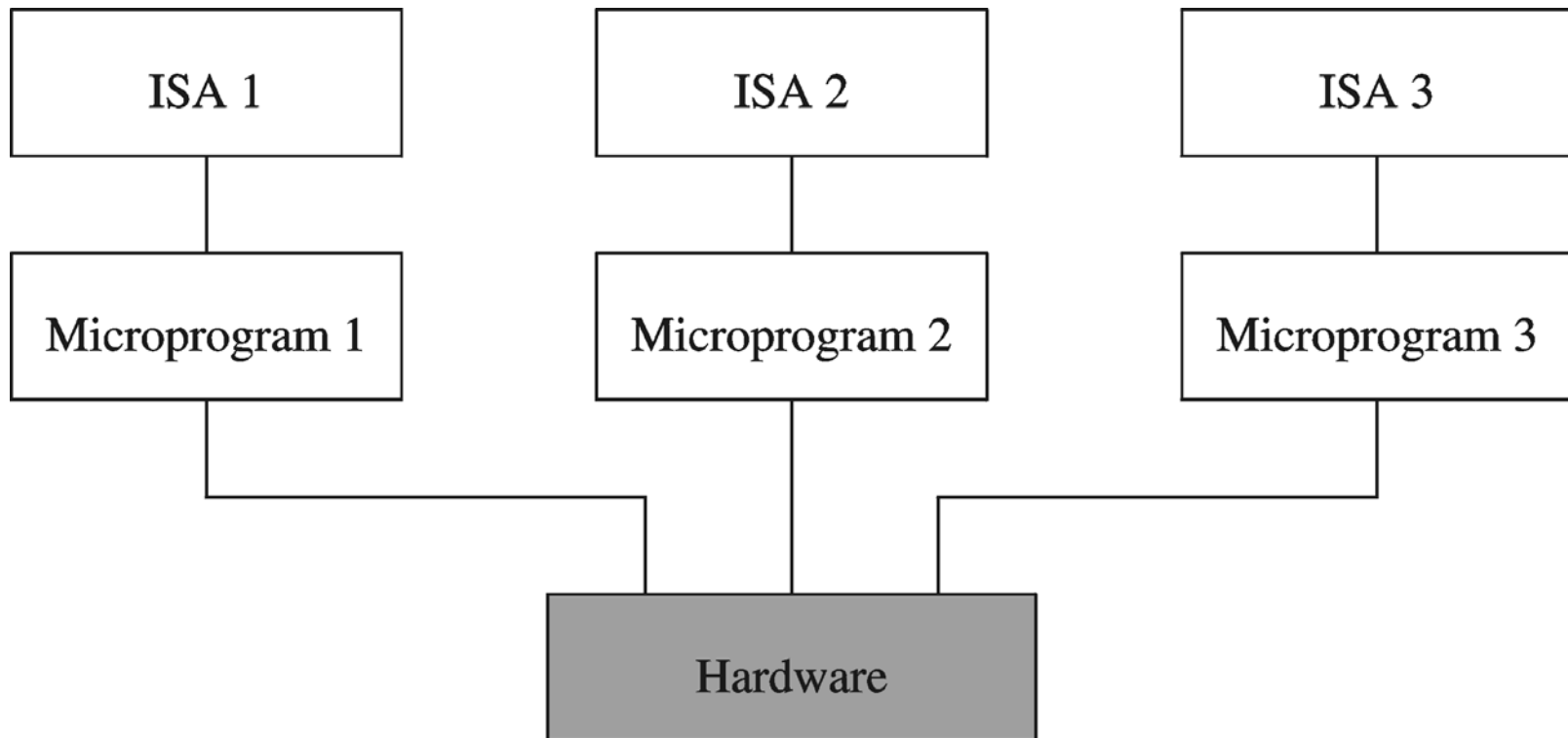


(b) RISC implementation

## Processor (cont'd)

---

- Variations of the ISA-level can be implemented by changing the microprogram



# Memory

---

- Ordered sequence of bytes
  - \* The sequence number is called the *memory address*
  - \* Byte addressable memory
    - » Each byte has a unique address
    - » Almost all processors support this
- Memory address space
  - \* Determined by the address bus width
  - \* Pentium has a 32-bit address bus
    - » address space = 4GB ( $2^{32}$ )
  - \* Itanium with 64-bit address bus supports
    - »  $2^{64}$  bytes of address space

# Memory (cont'd)

---

Address (in decimal)		Address (in hex)
$2^{32}-1$		FFFFFFFF
		FFFFFFFE
		FFFFFFFD
	• • •	
2		00000002
1		00000001
0		00000000

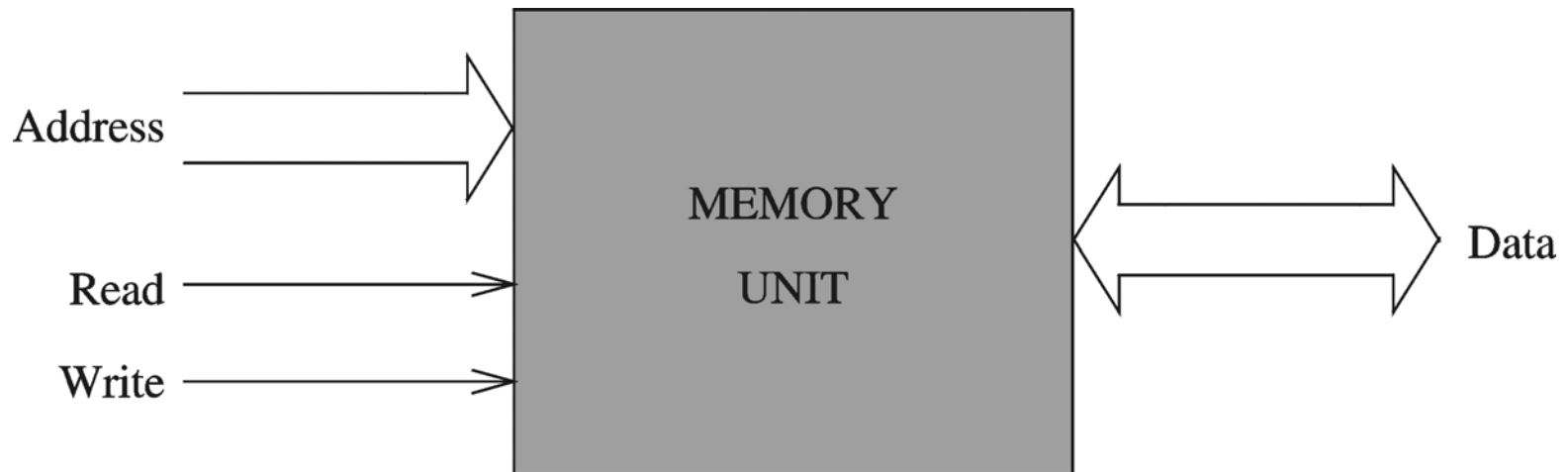
---



# Memory (cont'd)

---

- Memory unit
  - \* Address
  - \* Data
  - \* Control signals
    - » Read
    - » Write



## Memory (cont'd)

---

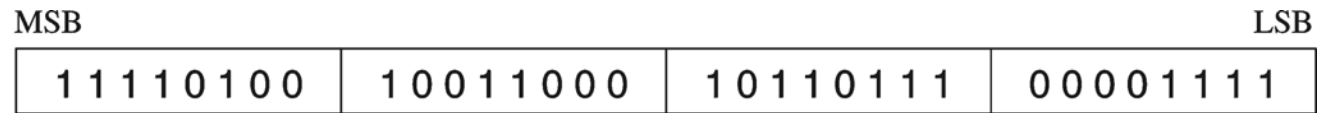
- Read cycle
  1. Place address on the address bus
  2. Assert memory read control signal
  3. Wait for the memory to retrieve the data
    - » Introduce *wait states* if using a slow memory
  4. Read the data from the data bus
  5. Drop the memory read signal
- In Pentium, a simple read takes three clocks cycles
  - » Clock 1: steps 1 and 2
  - » Clock 2: step 3
  - » Clock 3 : steps 4 and 5

## Memory (cont'd)

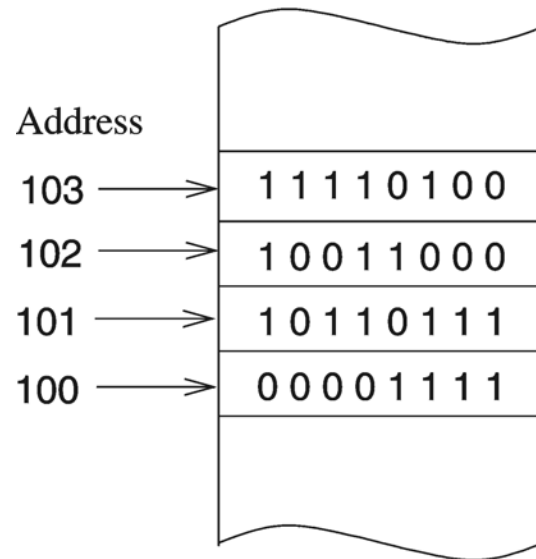
---

- Write cycle
  1. Place address on the address bus
  2. Place data on the data bus
  3. Assert memory write signal
  4. Wait for the memory to retrieve the data
    - » Introduce *wait states* if necessary
  5. Drop the memory write signal
- In Pentium, a simple write also takes three clocks
  - » Clock 1: steps 1 and 3
  - » Clock 2: step 2
  - » Clock 3 : steps 4 and 5

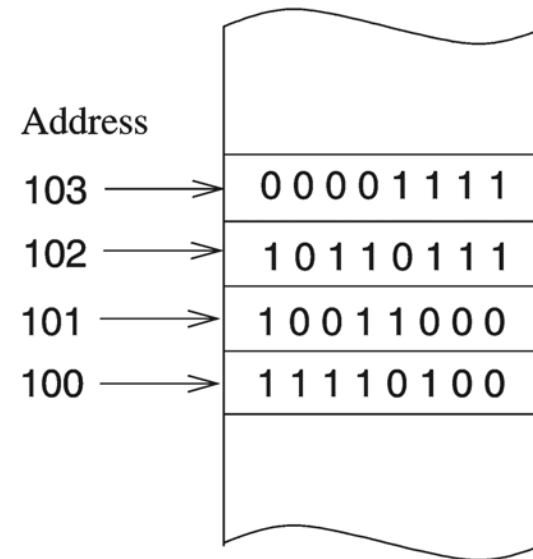
# Byte Ordering



(a) 32-bit data



(b) Little-endian byte ordering



(c) Big-endian byte ordering

## Byte Ordering (cont'd)

---

- Multibyte data address pointer is independent of the endianness
  - \* 100 in our example
- Little-endian
  - \* Used by Pentium
- Big-endian
  - \* Default in MIPS and PowerPC
- On modern processors
  - \* Configurable

# Design Issues

---

- Slower memories

Problem: Speed gap between processor and memory

Solution: Cache memory

- Use small amount of fast memory
- Make the slow memory appear faster
- Works due to “reference locality”

- Size limitations

- \* Limited amount of physical memory

- » Overlay technique

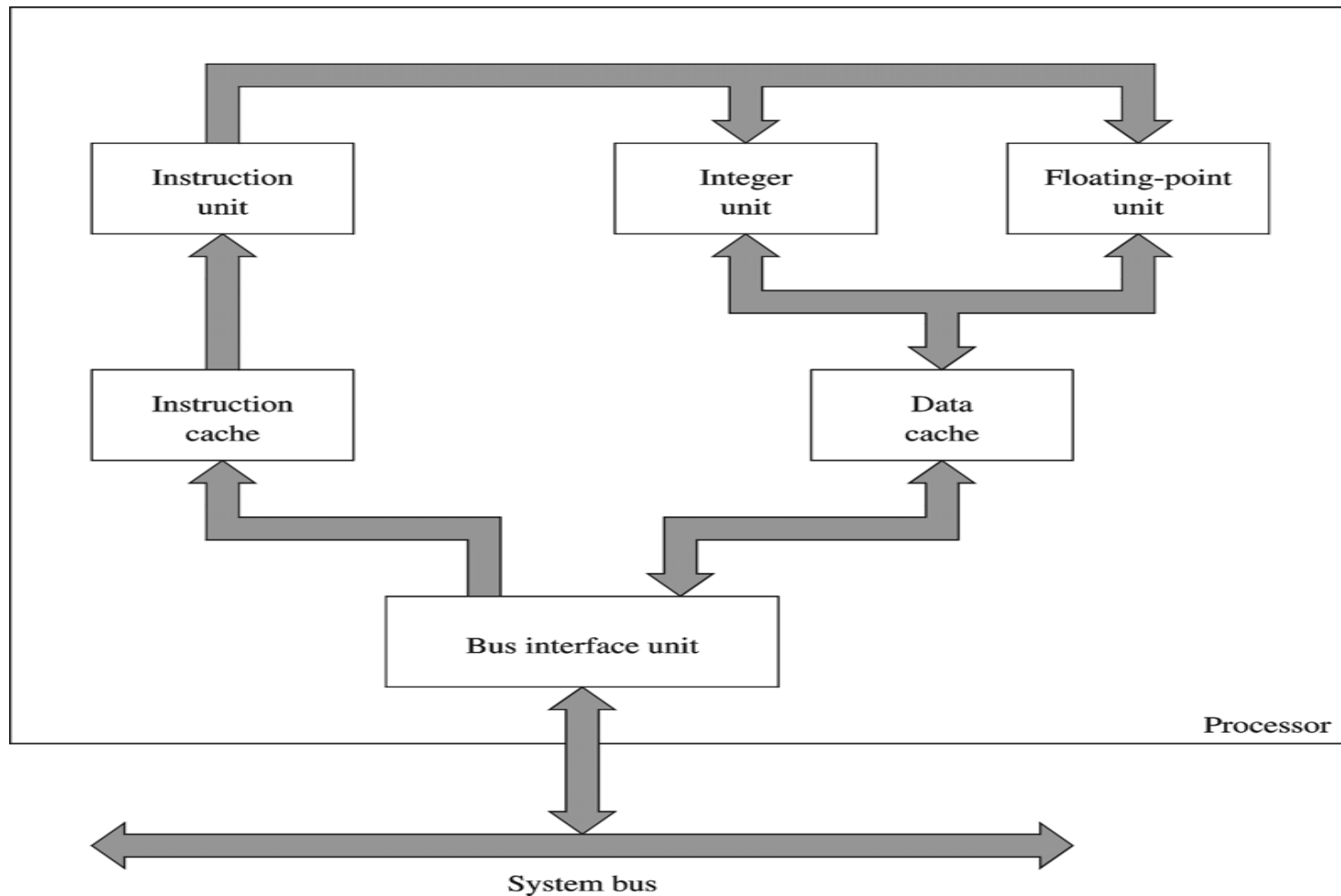
- Programmer managed

- \* Virtual memory

- » Automates overlay management

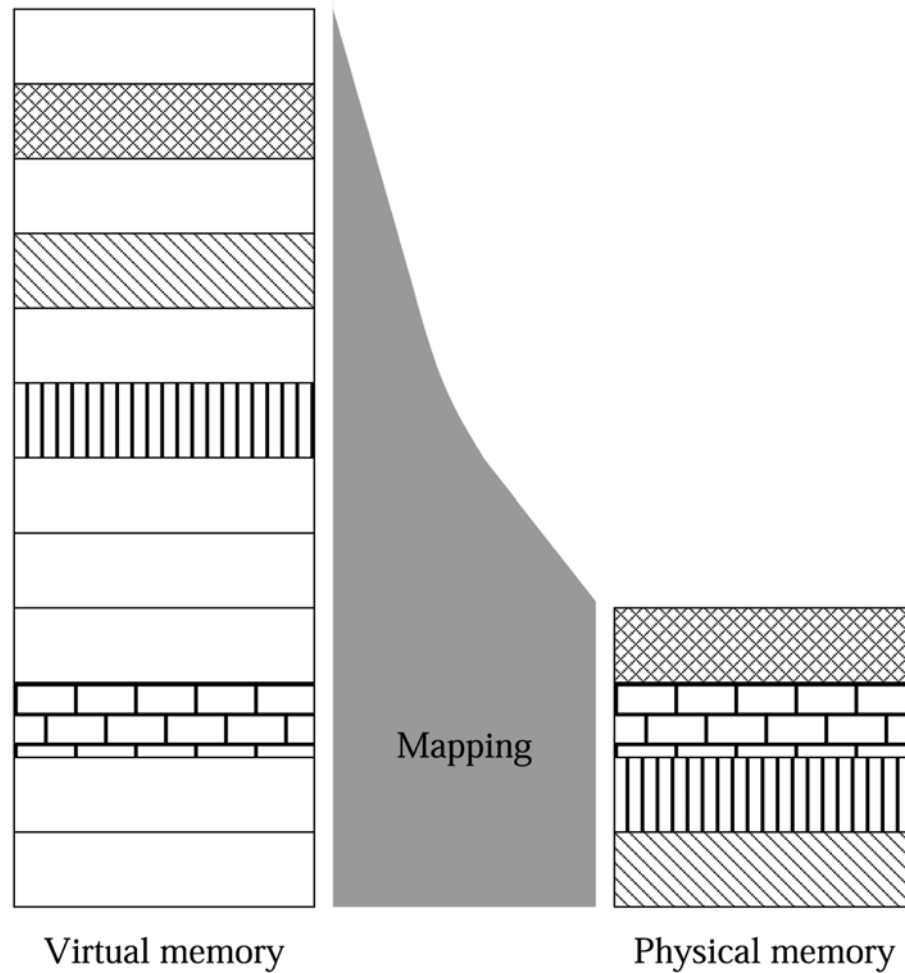
- » Some additional benefits

# Design Issues (cont'd)



# Design Issues (cont'd)

---





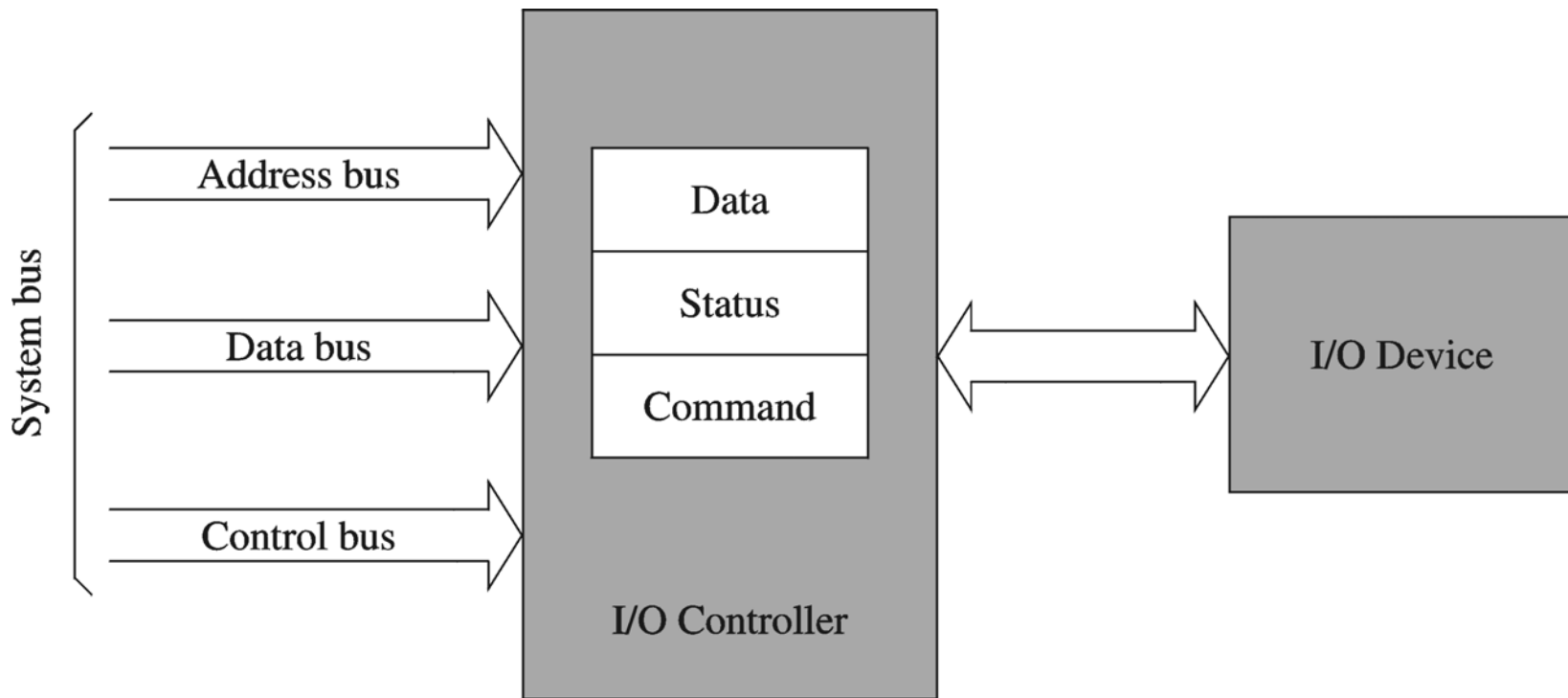
# Input/Output

---

- I/O devices are interfaced via an I/O controller
  - \* Takes care of low-level operations details
- Several ways of mapping I/O
  - \* Memory-mapped I/O
    - » Reading and writing similar to memory read/write
    - » Uses same memory read and write signals
    - » Most processors use this I/O mapping
  - \* Isolated I/O
    - » Separate I/O address space
    - » Separate I/O read and write signals are needed
    - » Pentium supports isolated I/O
      - Also supports memory-mapped I/O

# Input/Output (cont'd)

---



# Input/Output (cont'd)

---

- Several ways of transferring data
  - \* Programmed I/O
    - » Program uses a busy-wait loop
      - Anticipated transfer
  - \* Direct memory access (DMA)
    - » Special controller (DMA controller) handles data transfers
    - » Typically used for bulk data transfer
  - \* Interrupt-driven I/O
    - » Interrupts are used to initiate and/or terminate data transfers
      - Powerful technique
      - Handles unanticipated transfers

# Interconnection

---

- System components are interconnected by buses
  - \* Bus: a bunch of parallel wires
- Uses several buses at various levels
  - \* On-chip buses
    - » Buses to interconnect ALU and registers
      - A, B, and C buses in our example
    - » Data and address buses to connect on-chip caches
  - \* Internal buses
    - » PCI, AGP, PCMCIA
  - \* External buses
    - » Serial, parallel, USB, IEEE 1394 (FireWire)

# Interconnection (cont'd)

---

- Bus is a shared resource
  - \* Bus transactions
    - » Sequence of actions to complete a well-defined activity
    - » Involves a master and a slave
      - Memory read, memory write, I/O read, I/O write
  - \* Bus operations
    - » A bus transaction may perform one or more bus operations
      - Pentium burst read
        - Transfers four memory words
        - Bus transaction consists of four memory read operations
  - \* Bus arbitration

# Historical Perspective

---

- The early generations
  - \* Difference engine of Charles Babbage
- Vacuum tube generation
  - \* Around the 1940s and 1950s
- Transistor generation
  - \* Around the 1950s and 1960s
- IC generation
  - \* Around the 1960s and 1970s
- VLSI generation
  - \* Since the mid-1970s

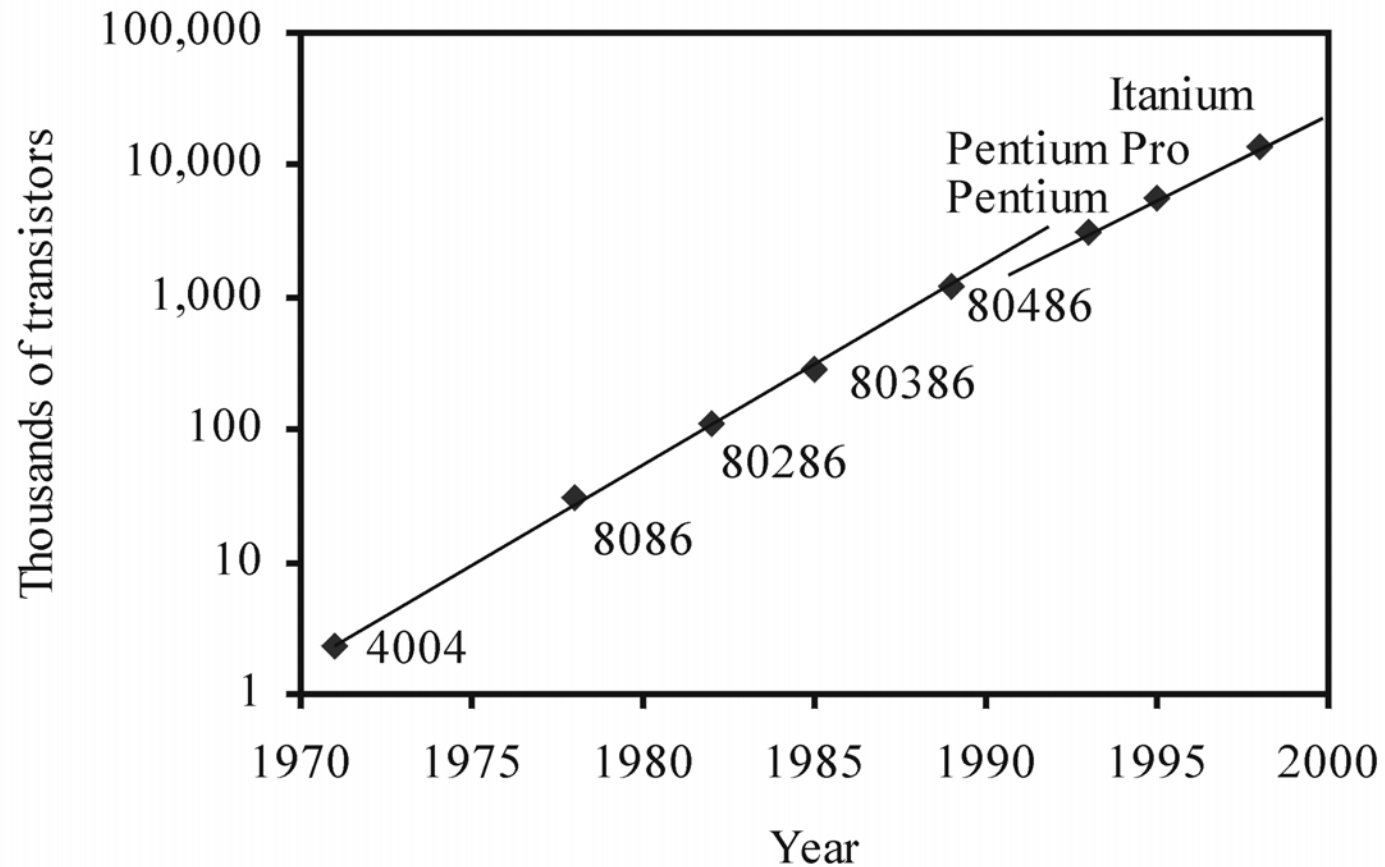
# Technological Advances

---

- Transistor density
  - \* Until 1990s, doubled every 18 to 24 months
  - \* Since then, doubling every 2.5 years
- Memory density
  - \* Until 1990s, quadrupled every 3 years
  - \* Since then, slowed down (4X in 5 years)
- Disk capacities
  - \* 3.5” form factor
  - \* 2.5” form factor
  - \* 1.8” form factor (e.g., portable USB-powered drives)

# Technological Advances (cont'd)

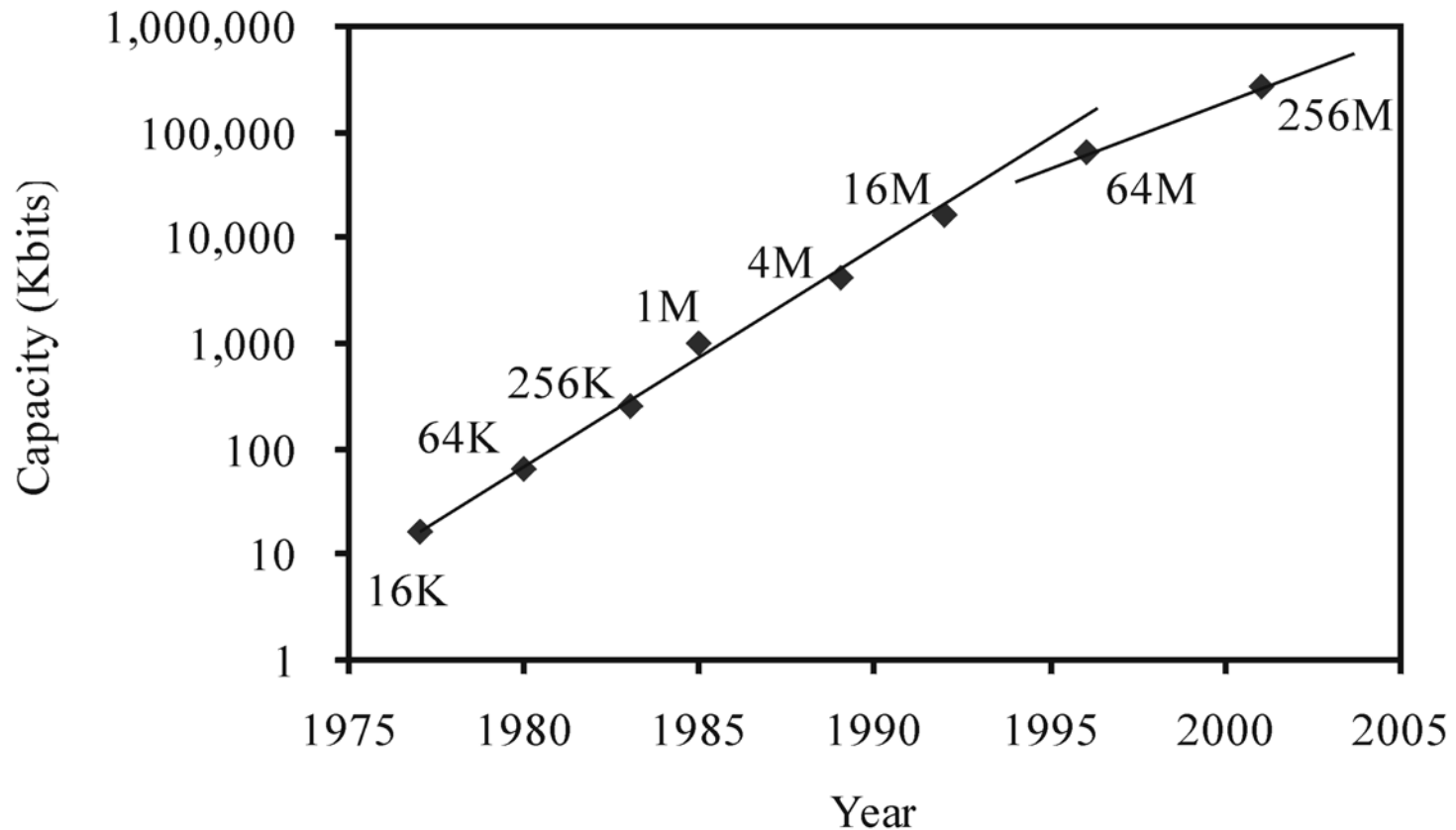
---





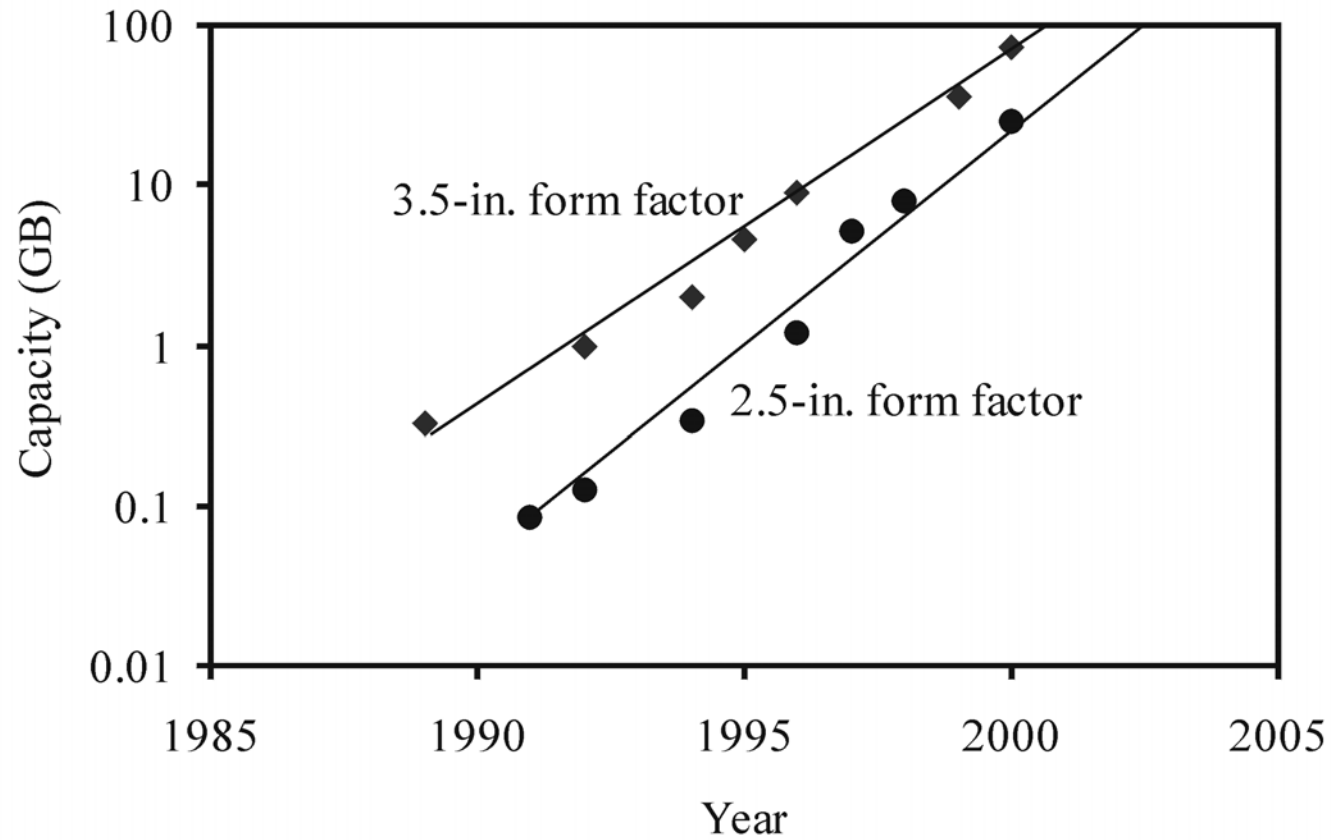
# Technological Advances (cont'd)

---



# Technological Advances (cont'd)

---



Last slide